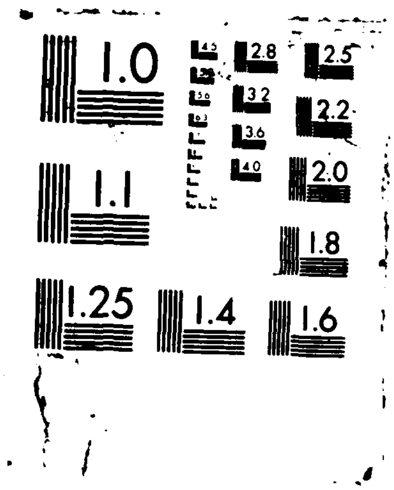
AD-A190 386 SOFTWARE STRUCTURING PRINCIPLES FOR VLSI CAD(U) 1/1
MASSACHUSETTS INST OF TECH CAMBRIDGE ARTIFICIAL
INTELLIGENCE LAB J KATZENELSON ET AL. 03 DEC 87
UNCLASSIFIED AI-M-1008 N00014-86-K-0180 F/G 12/5 NL


1.0
2.8
2.5
3.2
2.2
3.6
4.0
2.0
1.1
1.8
1.25
1.4
1.6

DTIC FILE COPY (4)

AD-A190 386

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

AI Memo 1008 — *December, 1987*

Software Structuring Principles for VLSI CAD

Jacob Katzenelson and Richard Zippel

Abstract

VLSI CAD systems are typically large and undergo frequent changes. A frustrating aspect of these changes is that so little of the old available programs can be reused. The reason is that it takes too much time and effort to find the reusable pieces and recast them for the new use.

We believe that such systems should be designed for reusability by anticipating change. Our thesis is that this goal can be achieved by designing the software as layers of problem oriented languages, which are implemented by suitably extending a "base" language. A language layer rarely needs to be adapted to changes, only the application (i.e. algorithm) needs to be changed.

We illustrate this methodology with respect to VLSI CAD programs and a particular language layer: a language for handling networks. A concept shared by many CAD programs is that of networks consisting of components and their interconnects. We capture this common part by providing a *language* for handling network problems. Such a language consists of our "base" language (EC or Lisp) plus data types, operations and control structures that are relevant to network problems. The network language is but one of several languages used; other languages we use deal with sets, two dimensional layout structures, waveforms, etc. The discussion of the network language illustrates this technique.

We present two different implementations of the above philosophy. The first uses UNIX and Enhanced C, a set oriented language supporting data abstractions based on C. The second approach uses Common Lisp on a Lisp machine. In each case, we describe the basic technique and its applications. We conclude by comparing the two approaches.

Keywords: Software reusability, Design of large software systems, Design by Language layers, Extended C, Lisp.

The first author is supported by the Technion Fund for the Encouragement of Research and the Steiner Fund at the Technion. While at the Massachusetts Institute of Technology his work was supported by DARPA contract N00014-86-K-0180. The work of the second author was supported by DARPA contract N00014-80-C-0622 while he was a member of the Electrical Engineering and Computer Science Department of the Massachusetts Institute of Technology.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER 1008	2. GOVT ACCESSION NO. ADA190386	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Software Structuring Principles for VLSI CAD		5. TYPE OF REPORT & PERIOD COVERED AI-Memo
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Jacob Katzenelson and Richard Zippel		8. CONTRACT OR GRANT NUMBER(s) N00014-86-K-0180 N00014-80-C-0622
9. PERFORMING ORGANIZATION NAME AND ADDRESS Artificial Intelligence Laboratory 545 Technology Square Cambridge, MA 02139		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Advanced Research Projects Agency 1400 Wilson Blvd. Arlington, VA 22209		12. REPORT DATE December 3, 1987
		13. NUMBER OF PAGES -16-
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) Office of Naval Research Information Systems Arlington, VA 22217		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE
16. DISTRIBUTION STATEMENT (of this Report) Distribution is unlimited.		
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)		
18. SUPPLEMENTARY NOTES None		
19. KEY WORDS (Continue on reverse side if necessary and identify by block number) Software reusability, Design of large software systems, Design by Language layers, Extended C, Lisp.		
20. ABSTRACT (Continue on reverse side if necessary and identify by block number) ail on reverse side.		

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE S/N 0102-014-6601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

Abstract

VLSI CAD systems are typically large and undergo frequent changes. A frustrating aspect of these changes is that so little of the old available programs can be reused. The reason is that it takes too much time and effort to find the reusable pieces and recast them for the new use.

We believe that such systems should be designed for reusability by anticipating change. Our thesis is that this goal can be achieved by designing the software as layers of problem oriented languages, which are implemented by suitably extending a "base" language. A language layer rarely needs to be adapted to changes, only the application (i.e. algorithm) needs to be changed.

We illustrate this methodology with respect to VLSI CAD programs and a particular language layer: a language for handling networks. A concept shared by many CAD programs is that of networks consisting of components and their interconnects. We capture this common part by providing a *language* for handling network problems. Such a language consists of our "base" language (EC or Lisp) plus data types, operations and control structures that are relevant to network problems. The network language is but one of several languages used; other languages we use deal with sets, two dimensional layout structures, waveforms, etc. The discussion of the network language illustrates this technique.

We present two different implementations of the above philosophy. The first uses UNIX and Enhanced C, a set oriented language supporting data abstractions based on C. The second approach uses Common Lisp on a Lisp machine. In each case, we describe the basic technique and its applications. We conclude by comparing the two approaches.

Software Structuring Principles for VLSI CAD

Jacob Katzenelson[1]
Department of Electrical Engineering
Technion—Israel Institute of Technology
Haifa, Israel 32000

Richard Zippel
Symbolics, Inc.
11 Cambridge Center
Cambridge, MA 02142

1. Introduction

CAD software systems are a collection of programs where the results of one program are the inputs of another. These programs are large, complex and usually CPU limited—often written independently and put together as the need arises. The programs are frequently changed. These changes may be due to changes in VLSI technology, but often they are also a result of the continuing evolution of the CAD algorithms themselves. A better understanding of an algorithm often leads to performance improvements or better use of the algorithm. These improvements may be conceptually simple while their effect is pervasive, such as changing a datatype from a list to a 2-D tree. As a result, a satisfying CAD system is a moving target.

A frustrating aspect of writing new systems is that so little of the old available programs can be reused. The reason is that it takes too much time and effort to find the reusable pieces and recast them for the new use. To reuse someone else's software requires understanding the details of his or her program. Sometimes understanding these unstructured details takes more effort than rewriting the program again.

We believe that such systems should be design for reusability by anticipating change. Our thesis is that this goal can be achieved by designing the software by layers of problem oriented languages. These languages are implemented by suitably extending a "base" language.[2]

In this paper we illustrate the above methodology with respect to VLSI CAD programs and a particular language layer: a language for handling networks. Networks made of components and their interconnections is a concept shared by many CAD programs. We capture that common part by providing a *language* for handling network problems. Such a language consists of our "base" language (EC or LISP) plus data types, operations and control structures that are relevant to network problems. The network language is but one of several languages used; other languages we used deal

[1] Currently on Sabbatical at the Artificial Intelligence Laboratory and Department of EECS at MIT, Cambridge, MA 02139

[2] Some of these problems cannot be eliminated as a result of the complexity of the algorithms themselves.

with sets, two dimensional layout structures, waveforms, etc. The discussion of the network language illustrates this technique.

A key part of the implementation technique is that the base language is extensible. We solve a particular set of problems by extending the base language, forming a *problem oriented language* for the particular problem at hand (for example, a language for manipulating waveforms or the network language discussed below). Of particular importance are language layers that capture concepts intrinsic to the field. Design based on such layers has the follow advantages:

(a) The programs are self documenting and document themselves concisely. The bulk of the documentation describes the problem oriented language and not individual programs.

(b) Changes involve writing a new algorithm in the same (problem oriented) language rather than writing a new language. This automatically ensures that new programs use parts of old programs. These parts are the concepts encapsulated in the language.

(c) The language rarely needs to be adapted to changes, only the application (i.e. algorithm) has to be changed. Occasionally, the language may also need to be extended.

We base our conviction in the power of the language approach upon the progress made in other fields by developing languages to express problems and their solutions. For example, one of Newton's major contribution to physics was the development of Calculus which can be viewed as a collection of new concepts and ways of putting them together.

The changes in digital design field over the last two decades have been so drastic that one wonders how the field maintained its unity. We claim that the stability of the practitioners language, the TTL conventions, is the source of this unity. In spite of the technological changes the language adapted remaining relatively constant, maintaining communication among workers in the field and preserving the field unity.

Consider another aspect of this same problem. Most conventional design methodologies begin with a *specification* of the system to be built. The specification should be as complete and as accurate as possible. Using this specification, the system is divided into smaller systems, each with their own specification. These sub-systems can be attacked independently since they need only satisfy their specifications. This methodology is generally known as *structured design* and is widely acclaimed and accepted.

We, however, feel that this methodology is inadequate for building large systems. The key flaw in structured design is the reliance on the accuracy of the specification. For sufficiently large systems, *the specification is always incorrect*. At the beginning of the project, the project itself is not completely understood and neither are its components. During the project, the better understanding the algorithms, data structures and their uses leads to modifications of the detailed specifications. At the of the project, what was built may meet the original specification, which was not what the client really had wanted. The client had not specified what he had thought he had specified.

How can we deal with incorrect and ever changing specifications? We can only assume that the evolving specification is not far wrong. It describes a family of systems near the desired one. We suggest building a platform on which this nearby family of systems can be built quickly. This platform is the problem oriented language we advocate. This language simplifies the construction of the specified system, and when the specification is modified, it is straightforward to build a new version of the system. The platform itself remains constant in the face of changes in the specification, since it depends upon the problem domain rather than a specific problem.

At this point an historical note is in order. In the LISP community, Joel Moses was the first to suggest that layered languages would be an effective way of dealing with design in the face of change. In the programming language community the related notions are "problem oriented languages" and "extensible languages." The practice of problem oriented languages leads to layers of languages. Conversely, the implementation of layers of languages leads uses language extensibility in order to build problem oriented languages.

We start by presenting the common denominator for network problems—the language for handling networks. We continue by presenting two different implementations of the above philosophy. The first uses UNIX and Enhanced C [8], a set oriented language supporting data abstractions based on C. The second approach uses Common Lisp on a LISP machine [13, 15]. In each case, we describe the basic technique and its applications. We conclude by comparing the two approaches. The two authors got together to write this article when they discovered their goals and philosophy were similar, but their techniques differ.

The body of the paper is organized as follows. Section 2 describes a general network framework for CAD that both the EC+UNIX and the LISP approaches share. This is the fundamental abstraction used in this paper. In Section 3, we discuss how EC can be used to implement and take advantage of the network abstraction. Section 4 briefly summarizes the different approaches used by LISP for the same problem. The methodology has been used for constructing logic simulators [7], VLSI layout system [17] and a system for analyzing linear networks. In the final section we discuss the practical experience gained by using the methodology and we also compare the two approaches taken.

2. Basic Network Model and Its Language

We speak about *entities* that have *endpoints*. Networks are formed by identifying (connecting) endpoints of entities. An entity has *properties* that contain values. One such property of an entity is its **type**. The value of the **type** property determines the number and kind of the other properties that the entity has. In network problems we have a type *node* and a type *component*. A node has a single endpoint that can only be connected to the endpoints of components. This is illustrated in figure **1**. Clearly, this is a very general model— perhaps more general than required for networks. We are not interested in minimal models, instead we show that the major network constructs

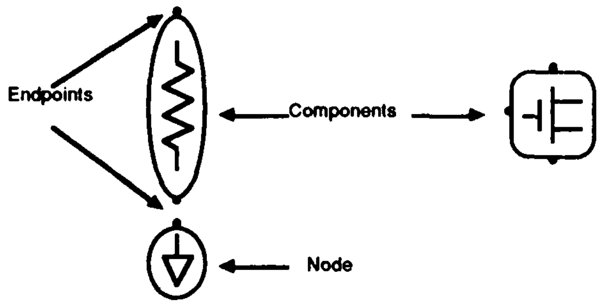


Figure 1. An example of Nodes and Components

can be built simply within this model.

A hierarchical network is created by using a component, C, with a property `parent` whose value is a network η. The nodes of η are assigned an `endpoint` property that is either "empty" or refers to a suitable endpoint of C.

A *homological copy* of a network is an important and useful concept [10]. A homological copy of η is a another network μ such that each node or component of η is associated with a node or component of μ (possibly a many to one relation). Clearly that association is a property of μ's elements and μ can be manipulated at will as long as η remains constant. As an example consider the schematics that a designer might use to describe a circuit. The schematics have more spatial detail than would be needed by a simulator which would work with a topological equivalent representation of the schematics. This topology is a homological copy of the schematics. This concept of homological copies is similar to the Sussman's Slices [16].

Around these entities we construct a language that enables us to handle these entities. The language allows us to refer to and change their properties; perform transformations on them and use the entities in more complex control structures. In more technical terms, we define a language by defining constructors, selectors, various other operations, and control abstractions that are appropriate to our network problems. The following examples illustrate both the language and its use for writing CAD application programs.

2.1 Displaying a Circuit

Each element of the network (both nodes and components) has a property, say `picture`, whose value is (refers to) the code that displays the element picture on the screen. In addition, there is a `coordinates` property that give the location of the element on the screen. The following loop generates a picture of the network:

```
forall elements e of network N do
    display_element(e)
```

`display` checks if `e.picture` is `nil`; if so, it recurses on `e`'s subcircuit.

```
display_element(element) {
    if (element.picture == nil)
        display_network(element.network)
    else
        element.picture(element.coordinates)
    }
```

Obviously, the same sort of operations could be written in LISP. Throughout this paper all of our LISP programs are written in Common Lisp [15]. We would begin by writing a function that takes two arguments. The first is a network and the second is a functional argument that is applied to each element of the first argument. Call this function `map-over-elements`.

```
(defun display-network (network)
  (map-over-elements network
    (function display-element)))
```

The `display-network` function displays a network by applying the function `display-element` to each element of `network`. The form `(function display-network)` is returns the function associated with the identifier `display-element`. This extra bit of complexity is needed because Common Lisp allows a value and a function to be associated with the same symbol.

```
(defun display-element (element)
  (if (null (picture element))
      (display-network (network element))
      (funcall (picture element) (coordinates element))))
```

2.2 Simulation

Consider the following examples of simulation programs: synchronous and asynchronous logic simulation, analog simulation, timing simulation [9], etc. Although quite different, these programs have much in common. Each can be generated by programs that traverse the networks, using the properties of the network elements. In each case the handling of the results is similar; result waveforms have to be associated with the network elements so that the user can refer to them as properties of the elements.

For each kind of simulation we need different network element properties and a different "generation function"—the function that generates the simulator from the network description.

The above is illustrated by an example of a simple synchronous logic simulation. We assume that the behavior of each component, e, is a finite state machine described by

$$\vec{s}_{n+1} = f_e(\vec{s}_n, \vec{I}_n),$$

where $\vec{s}_n$ is the state vector at time n and $\vec{I}_n$ is the input vector at time n. For simplicity assume that $\vec{s}_n$ is also the output vector at time n and that inputs to all components are outputs of other components. f_e is called the *device function*.

The following simple program generates the simulation program.

```
forall elements e of network N do {
    gen( state and next_state variable declarations for e) ;
    gen("get_initial_conditions(e.state)" ) ; }
gen ("for (t = t0 ; t <= t max ; t++) {");
forall elements e of network N do {
    gen ("e.devicefun(e.inputs, e.state, e.next_state);" ) ; }
gen ("forall elements e of network N do {
                              e.state = e.next_state;
                              store_results(e.state);}"  );
gen ("}");
```

The following LISP fragment implements the same functionality as the previous EC one. In LISP we do not generate a program but we run directly off the data structure. Notice also that we do not generate any variable declarations.

```
(map-over-elements N
  (lambda (e)
    (setf (state e) (initial-conditions e))))
(loop for t = t0 below t max
      do (map-over-elements N
           (lambda (e)
             (push (funcall (devicefun e) (inputs e) (state e))
                   (state-history e)))))
(loop for t = t0 below t max
      do (map-over-elements N
           (lambda (e)
      (setf (state e) (first (state-history e))))))
```

The special forms begining with `lambda` are used to create anonymous functions. The special form `setf` is used to perform all assignments in Common Lisp. The `state-history` slot of each element is a list of the states through which the element has passed. The first component of `state-history` corresponds to the `next-state` variable used in the EC program.

What these two approaches have in common is that each element must contain its device function, information about its state, initial conditions and connectivity. The following remarks relate to these two versions of the simulation example.

(a) The example is indeed straight forward. Some of the simplicity is a result of this particular simulation method. Under the conditions stated, the order of the calls to the device function in the time loop can be arbitrary.
However, we believe that given the appropriate network language the "core" of the any network application is quite simple. That is, it is simple compared with the layers of software around it - the I/O package, the handling of libraries, the manipulation of results, etc. In our case, these facilities are provided as part of the software environment. They are provided as standard interfaces (driven by sub-languages) to be used by any network application. The internal complexities of these facilities are the user.

(b) The EC version generates a program. This program is compiled and run to produce results. The LISP version is a program that runs to produce results. The techniques and differences between the two approaches are elaborated in the sequel.

(c) With each application, different information is associated with the network elements.

(d) In all cases the computed results have to be associated with the original network model to enable communications with the user.

3. Implementation using Enhanced C plus UNIX

The design of Enhanced C (EC) and its philosophy are described in [8]. EC is a set-oriented, extensible, C-like language. Extensibility in EC involves the use of data abstractions to define new types. EC's data abstractions, called clusters, are macro-like devices that perform substitutions on the typed syntax tree. EC's set orientation and data abstractions are important to our discussion. We shall elaborate on these topics before describing the organization of the CAD system.

A set-oriented language is a language containing high level data aggregates. In EC these are sets, sequences, tuples and "oneof's." The semantics of sets and sequences are as in mathematics. As programming entities they may be thought of as the generalization of the array. Tuples and oneofs are the generalization of C's "structure" and "union" respectively. With these data aggregates come expressions and control statements that resemble set usage in mathematics; for example:

Let S be a set of a given type, and x an object the same type.

```
add x to S;
exist x in S suchthat < predicate>
forall x in S suchthat < predicate> do < statement>
```

Set orientation helps program combinatorial problems, a type of problems common in VLSI CAD. It provides a concise notation that, since it is part of our common education, helps in both programming and documentation.

Data abstractions are means of declaring new types. The following few lines of code is an example of such a declaration.

```
cluster list (T)                                                /* EC 1 */
mode  T; where int T;                                           /* EC 2 */
{
/* This list implements a set of elements of type T storing copies of
   elements in the list */

rep type list_st *l;                                            /* EC 3 */
typedef struct {  type list_st *next;
                  type T member;
               } list_st;

fun int doit(...)...{ ... }                                     /* EC 4 */


oper void oper forall(elment,llist,st)                          /* EC 5 */
type T elment;
type list llist;
statement st;
{
 type l p;
  for(p=llist; p!=(type l)nil; p=p->next)
   { elment = (p->member);
     st;
   }
}

proc int oper add(elment,llist)                                 /* EC 6 */
type T elment; type list llist;
{
  type l p;    /* declaration */
  for (p = (type l) malloc(sizeof(type list_st));
  p->member = elment;
  p->next = llist;
  llist =p;
  result l;
}


oper type list oper +(...) ... { ... }                          /* EC 7 */

constant type list oper null(cluster_name)                      /* EC 8 */
cmode cluster_name;                                             /* EC 9 */
{
 (type l)nil
}

/* end cluster list */
}
```

A type is defined by its representation (EC 3) the data structure of a variable of this type; and by a set of operations. Line EC 4 defines the operation `forall` whose call is as follows:

```
forall element in llist do  statement
```

Line EC 6 defines the operation `add`, which may be used in the following manner:

```
add element to llist
```

Line EC 7 overloads "+" to be the union of two sets. Line EC 6 defines a constant, `null`, the empty set.

The cluster operations are selected by their name and the types of their arguments (polymorphic operations), e.g. "+" may denote the addition of integers, the union of sets, etc. The compiler selects the appropriate operation among those built-in and those defined by the user.

The EC clusters are parameterized. E.g. `T` on line EC 1 is a type: this cluster defines a set of `T`. EC cluster operations can be implemented by either procedures or *macros*. The former case is illustrated on line EC 6 (key word `proc`) and the later is illustrated on line EC 5. The macros perform substitutions on the typed syntax tree.

Macro implementations of cluster operations have several interesting properties. They avoid the procedure call overhead and, when the body of the operation is small enough they produce code that may be both faster and smaller. Since binding is done at compile time they accept both statements and types as arguments and bind variables "by name." They may return "lvalues" and appear on the left hand side of the assignment statement.

The following sections describe the general structure of the CAD system and then dwell on the language aspects that are central to our discussion.

3.1 System Organization

Figure **2** summarizes the system organization as a set of data structures with the transformations from one data structure to another. The transformations are represented as arrows.

The input is described by an input language. The components are described by giving a (possibly parameterized) type and a name. For example,

```
AND(WIRE(2)) a1, a2;
```

describes a_1 and a_2 as two components of type `AND(Wire(2))`. `Wire(2)` is a type and is used as a parameter of type `AND`. A component's connections are described by stating

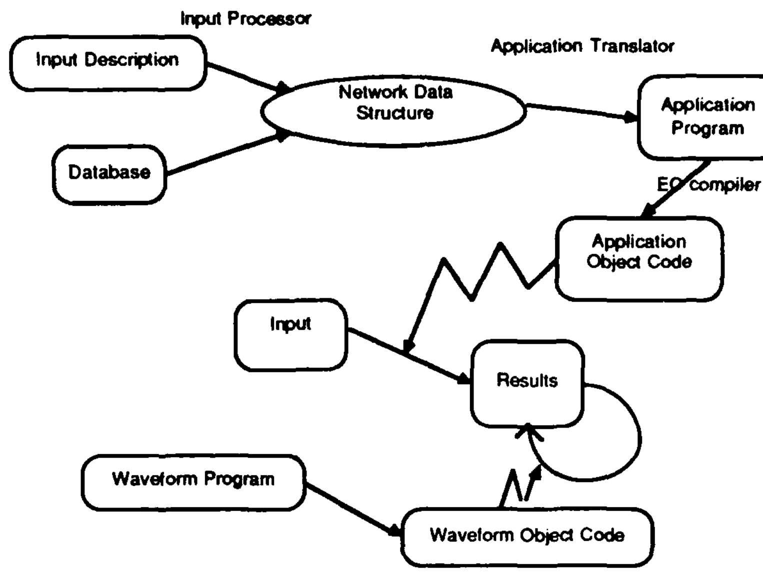


Figure 2. CAD tools using EC

which endpoints are connected to which component's input endpoint. Although this example does not show it, the input usually describes nested networks, i.e., components are defined by means of networks.

The input language is processed by a program that we call the *input processor*. When that program encounters a type it searches the *database* for an appropriate type description described below.

The database is arranged as Unix files and directories. A directory is devoted to each type (the *type's directory*). Each such directory contains a file describing the component topology, i.e. names of endpoints, nature of endpoints (input, output, tri-state). For each application the type directory contains a file describing the component's properties, e.g. power and size, and a file containing the cluster describing the behavior of the type for each application.

The input processor uses the topology file for checking for input consistency and builds a general *network data structure* describing the network (the network data structure in figure **2**). Each element is described by an object (tuple) whose properties (components) are the basic characteristics of the elements (name, endpoints) and the application properties read from the application file.

The *application translator* is a program that runs on the general data structure and produces the *application program*. The application program is then translated by the EC compiler into runnable code, using data abstractions called from the database. The object code is run on the *input* to produce *results*. The (new) results are manipulated by a *waveform program* to produce other results that the user can view, print, or otherwise inspect.

For introducing a new application the user must provide the database entries for

the application and, in particular, write the application translator. All other facilities are provided by the system.

With EC as the main tool the language used to write an application translator is no mystery: it is EC proper plus a network cluster and clusters for implementing sets, sequences and tuples. Since the input usually defines nested networks, the network cluster contains operations that produce homological copies of nested networks. For sets, sequences, etc. the programmer uses the language facilities provided by EC and clusters he either extracts from the standard library or writes himself. The network cluster requires some additional elaboration.

Our network elements have two kinds of properties:

- properties that are application *independent* and are used to capture the nature for the elements as part of a network, e.g. connectivity. We call these properties network generic properties.
- Properties that are application dependent, e.g. the device function in the simulation application.

We would like to generate one cluster whose representation and operations include the generic operations and the application oriented properties. This is an "inheritance" mechanism — the new cluster inherits the properties of the network cluster and a (pseudo) cluster whose operations are the application dependent operations.

At this time, EC does not have automatic inheritance [4, 3, 14] and this particular inheritance is done in the way described below.

(1) The input processor generates the network data structure as a fixed part per component with a "hook" for application dependent properties. Each application dependent entry carries its name, its type and its value.

(2) The cluster that contains the generic network operation is augmented by operations that access the application properties. These operations are picked up from the database and inserted ("`include`" style) into the network cluster.

4. Use of Lisp Machines for CAD

SCHEMA was developed using Common Lisp on a Lisp machine. LISP is extremely well suited for building new languages. Several of the features of LISP contribute this property. Significant power is achieved through the use of data abstraction (as embodied by flavors [4], which includes inheritance). The free variables of procedures and functions can be closed over to create *closures*. These closures are first class citizens and can be passed to and from other functions and stored in data structures. Finally, its uniform syntax (lack of it) simplifies the creation of semantic extensions. The use of macros allows these extensions to be incorporated smoothly.

At least two integrated VLSI CAD systems have been built on Lisp machines. SCHEMA [6] is being developed at MIT with the help and collaboration of colleagues at Harris, Corp., and NS [5] which was developed at Symbolics, Inc. These two systems share a common heritage from the Daedalus system developed at MIT [2] and many key ideas.

SCHEMA was developed to handle all of the aspects of designing VLSI circuits. Thus it contains tools for schematic entry, layout and simulation. Furthermore, it is built upon a large body of tools that can be used to build additional CAD tools. Because the entire design and all of the tools reside in a single address space a somewhat different approach needs to be taken than is used in a Unix environment. And yet, there are still strong similarities with the approach used in EC. The following diagram corresponds to figure **2** used to describe the EC organization.

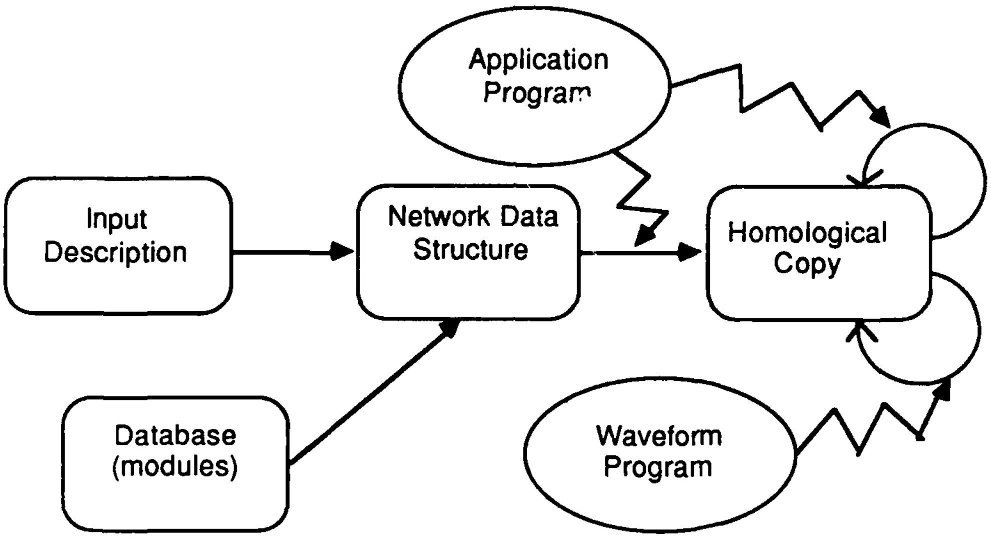


Figure 3. CAD tools using Schema

As with EC, the core of the design system is the network description language. In SCHEMA this language is interpreted as a procedure for generating a memory resident data structure that describes the topology of a circuit. In Figure **3** this is the "Input Description." This language may make use of definitions of other circuits from a library, thus allowing the hierarchical description of circuit.

The data structure that results is not totally passive. Code can be attached to components and nodes in the data structure. (This is the main use of objected oriented programming in SCHEMA.) Some of the code is included in the input description while other code is added by application programs. Thus a simulator might add code to model the voltage and current constraints of a particular device.

The hierarchical network data structure is not optimal for simulation purposes. For certain types of simulations it contains too much detail (logical simulators are not concerned with the construction of an AND gate) while for other simulation types the hierarchical structure gets in the way. To deal with both of these problems a homological copy is built which has the optimal structure for the application program. SCHEMA provides mechanisms for building and operating on the homological copies, which are extensions of the network description language.

In SCHEMA both the network data structure and its homological copies reside in memory simultaneously. The user tends to deal with the network data structure and is often not aware of the homological copy. To ease the use of the system, links are preserved between the homological copy and the network data structure. This allows questions about the result of simulations to be directed through the network data structure to the homological copies.

Perhaps the feature of the Lisp Machine environment that distinguishes it the most from Unix-like environments is that all the design tools, as well as the design itself, reside in a common address space. This characteristic makes it especially easy for disparate CAD tools to communicate through shared data structures. For instance, a schematic editor and a simulator are able to examine and modify a common data structure that represents a topology thus simplifying both the user interface to the simulator and the creation of incremental analysis tools. Far less programmer time is spent flattening data structures into text files and reparsing them for use in different tools. The "language" in which the CAD tools are built can include higher level primitives than text files. Furthermore, the co-residence of the different tools allows one CAD tool to invoke another when information is lacking. Thus a circuit simulator could invoke the layout program to inquire about the capacitances of different nodes. This approach allows the tools to interact as collection of cooperating experts rather than a sequence of files transducers.

The major disadvantage of this approach is that modifications and annotations to data structures persist until the next time the workstation is rebooted. This means that modifications to data structures must either be able to be undone when necessary or are permanent.

5. Practice and Experience

Two applications were built with EC using this philosophy. The first is a simulator for logic networks [7] that makes extensive use of the network layer mentioned earlier. The second system does automatic layout in a fashion similar to PI [17, 12]. The same network language layer was used to create nested networks, while most of the layout algorithms were built on another layer consisting of EC, sets and sequences. This system is intended as a testbed for a variety of different algorithms. The use of layered languages made the expressions of the algorithms concise, readable and easy to modify.

The network language layer of SCHEMA was used to build a variety of different simulators and as an interface to conventional circuit simulators like Spice [11]. In fact a very large number of languages layers were used in SCHEMA. Among them were layers for graphic imaging, for building user interfaces, for editing graphical structures, for dealing with permanent version of the data structures in SCHEMA, for manipulating symbolic mathematical expressions, etc. In each case, less experienced programmers were able to construct rather sophisticated applications using the linguistic tools provided.

6. Summary and Conclusions

The two systems share a common philosophy: To construct a CAD tool, first define a language that captures the concepts common to the field and widely accepted by workers in the field. Next, build the CAD tool using this language. For example, consider building a logic simulator. This simulator is an example of a class of programs that manipulate networks. We begin with a network language, and then extend it to one that is appropriate for constructing simulators. The logic simulator is written in the simulation language. Notice that we build in "extra" generality. The network language can be used for problems other than simulation. The simulation language can be used for either logic simulation or circuit simulation.

This philosophy is implemented in a similar fashion in both systems. In both cases, the system is built on a base language, EC and LISP respectively, with addition problem specific languages constructed by extending the base languages. In EC this is done using clusters, where the macro oriented operations are used for implementing control abstractions. In LISP this is done using flavors to implement data abstraction; the control abstractions are implemented using macros and functional arguments.

The differences between these two systems revolve around two points: the large address space used by LISP and the early binding by compilation used by EC. The Lisp machine world uses a very large address space allowing many tools and their data structures to co-exist, while the Unix/EC world places each tool in its own address space and shares data through the file system. The Unix/EC world tends to bind early have the compiler make as many decisions as possible, while the Lisp world delays those decisions until run time.

The large address space allows SCHEMA to rely on "in core" data structures rather than using files. This allows CAD tools to share common data structures and to communicate simply by passing pointers to these data structures. This environment allows the tools to appear as an single program. This eliminates the need to translate to and from characters to the data structures used by the CAD tools. Furthermore, CAD tools are now able to invoke other CAD tools more easily in this common environment.

The SCHEMA tools include the schematic entry system, the layout editor, simulator and others. These tools share common network data structures, but often have slightly different structures for their own use. These tools and their data structures all co-exist in the same large address space.

The early binding by compilation used by EC has advantages and disadvantages. The disadvantages are that the communication between the results and the original specification of the network has to be done via files and dictionaries, which complicates the CAD system. And if the application requires iteration across the boundaries between CAD tools that application becomes quite expensive. The advantage of early binding by compilation is the gain in run-time performance for CPU bounded problems.

The comparison between run-time performance of conventional machines running compiled EC and the Lisp machine running Lisp is not simple. We do not have significant tests that run in both environments. The Lisp system attains speed improvements

by special hardware and by compilation. We don't feel that there is any reason why this method is should be inherently slower or faster than a conventional environment.

Currently a LISP machine system is a relatively expensive item. The Unix system runs on conventional machines.

7. Acknowledgments

The first author is supported by the Technion Fund for the Encouragement of Research and the Steiner Fund at the Technion. While at MIT his work was supported by DARPA contract N00014 86 K-0180. The work of the second author was supported by DARPA contract N00014 80-C-0622 while he was a member of the Electrical Engineering and Computer Science Department of the Massachusetts Institute of Technology.

8. References

1. H. Abelson and G. J. Sussman, with J. Sussman, *Structure and Interpretation of Computer Programs*, MIT Press, Cambridge, MA, (1984).
2. J. Batali and A. Hartheimer, *The Design Procedure Language Manual*, MIT Artificial Intelligence Laboratory, (1980) .
3. G. M. Birtwistle, O. J. Dahl, B. Myhrhand and K. Nygaard, *Simula Begin*. Auerbach, Philadelphia, PA, (1973).
4. D.G. Bobrow, L. G. DeMichiel, R. P. Gabriel, K. Kahn, S. E. Keene, G. Kiczales, L. Masinter, D. A. Moon, M. Stefik and D. L. Weinreb, "Common Lisp Object System Specification," *Submitted to the X3J13 Common Lisp committee*, (1987).
5. J. Cherry, H. E. Shrobe, N. Mayle, C. Baker, H. Minsky, K. Reti, "NS: An Integrated Symbolic Design System," *VLSI '85: VLSI Design of Digital Systems*. Elsevier Science Publishers B. V., (1986).
6. G. C. Clark, R. Zippel, "Schema: An Architecture for Knowledge Based CAD," *Proceedings of ICCAD'85*, (1985), 50–52.
7. J. Hershtik, *A Generalized Programming Environment for Network Problems*, M.Sc. thesis, Dept. of Depart. of Electrical Engineering, Technion, (1987).
8. J. Katzenelson, "Introduction to Enhanced C (EC)," *Software Practice and Experience*, **13**, June, 1983, 551–576.
9. T. McWilliams, "Verification of Timing Constraints in Large Digital Systems," *Proceedings of the 17th Design Automation Conference*, (1980), 139–147.
10. E. Milgrom, *AEPL – An Extensible Programming Language*, Ph.D. thesis, Dept. of Electrical Engineering, Technion – Israel Institute of Technology, (1971).
11. L. W. Nagel, *SPICE2: A Computer Program to Simulate Semiconductor Circuits*, ERL-M520, University of California, Berkeley, (1975) .
12. R. Rivest, "The PI (Placement and Interconnect) System," *Proceedings of 19th Design Automation Conference*, (1982).
13. D. A. Moon, R. M. Stallman, D. L. Weinreb, *LISP Machine Manual, Fifth Edition*. MIT Artificial Intelligence Lab., Cambridge, MA, January, 1983.

14. D. Moon and S. Keene, "Object Oriented Programming with Flavors," *Proceedings of OOPSLA'86*, September, 1986.
15. G. L. Steele, Jr., *Common Lisp: The Language*, Digital Press, (1984).
16. G. J. Sussman, "SLICES: At the Boundary between Analysis and Synthesis," *Proc. Artificial Intelligence and Pattern Recognition in Computer Aided Design*, IFIP WG 5.2, Grenoble, France, (1978).
17. A. Turgeman, *Automated Integrated Circuits Layout*, M.Sc. thesis, Dept. of Electrical Engineering, Technion—Israel Institute of Technology, (1987).

END
DATE
FILMED
4-88
DTIC